

11 | 一枝独秀的字符串：C++也能处理文本？

2020-05-30 罗剑锋

罗剑锋的C++实战笔记

[进入课程 >](#)



讲述：Chrono

时长 15:32 大小 14.23M



你好，我是 Chrono。

在第一个单元里，我们学习了 C++ 的生命周期和编程范式。在第二个单元里，我们学习了自动类型推导、智能指针、lambda 表达式等特性。今天，我们又要开始进入一个新的单元了，这就是 C++ 标准库。

以前，“C++”这个词还只是指编程语言，但是现在，“C++”早已变成了一个更大的概念——不单是词汇、语法，还必须要加上完备工整的标准库。只有语言、标准库“双壁”，才能算是真正的 C++。反过来说，如果只单纯用语言，拒绝标准库，那就成了“天残地缺”。

看一下官方发布的标准文档吧（C++14），全文有 1300 多页，而语言特性只有 400 出头，不足三分之一，其余的篇幅全是在讲标准库，可见它的份量有多重。

而且，按照标准委员会的意思，今后 C++ 也会更侧重于扩充库而不是扩充语言，所以将来标准库的地位还会不断上升。

C++ 标准库非常庞大，里面有各式各样的精巧工具，可谓是“琳琅满目”。但是，正是因为它的庞大，很多人在学习标准库时会感觉无从下手，找不到学习的“突破口”。

今天我就先来讲和空气、水一样，最常用，也是最容易被忽视的字符串，看看在 C++ 里该怎么处理文本数据。

认识字符串

对于 C++ 里的字符串类 `string`，你可能最熟悉不过了，几乎是天天用。但你知道吗？`string` 其实并不是一个“真正的类型”，而是模板类 `basic_string` 的特化形式，是一个 `typedef`：

```
1 using string = std::basic_string<char>; // string其实是一个类型别名
```

 复制代码

这个特化是什么意思呢？

所谓的字符串，就是字符的序列。字符是人类语言、文字的计算机表示，而人类语言、文字又有很多种，相应的编码方式也有很多种。所以，C++ 就为字符串设计出了模板类 `basic_string`，再用模板来搭配不同的字符类型，就能够更有“弹性”地处理各种文字了。

说到字符和编码，就不能不提到 Unicode，它的目标是用一种编码方式统一处理人类语言文字，使用 32 位（4 个字节）来保证能够容纳过去或者将来所有的文字。

但这就与 C++ 产生了矛盾。因为 C++ 的字符串源自 C，而 C 里的字符都是单字节的 `char` 类型，无法支持 Unicode。

为了解决这个问题，C++ 就又新增了几种字符类型。C++98 定义了 `wchar_t`，到了 C++11，为了适配 UTF-16、UTF-32，又多了 `char16_t`、`char32_t`。于是，`basic_string` 在模板参数里换上这些字符类型之后，就可以适应不同的编码方式了。

 复制代码

```
1 using wstring = std::basic_string<wchar_t>;
2 using u16string = std::basic_string<char16_t>;
3 using u32string = std::basic_string<char32_t>;
```

不过在我看来，虽然 C++ 做了这些努力，但其实收效并不大。因为字符编码和国际化的问题实在是太复杂了，仅有这几个基本的字符串类型根本不够，而 C++ 一直没有提供处理编码的配套工具，我们只能“自己造轮子”，用不好反而会把编码搞得一团糟。

这就导致 `wstring` 等新字符串基本上没人用，大多数程序员为了不“自找麻烦”，还是选择最基本的 `string`。万幸的是 Unicode 还有一个 UTF-8 编码方式，与单字节的 `char` 完全兼容，用 `string` 也足以适应大多数的应用场合。


所以，我也建议你只用 `string`，而且在涉及 Unicode、编码转换的时候，尽量不要用 C++，目前它还不太擅长做这种工作，可能还是改用其他语言来处理更好。接下来，我就讲一讲，该怎么用好 `String`。

用好字符串

`string` 在 C++ 标准库里的身份也是比较特殊，虽然批评它的声音有不少，比如接口复杂、成本略高，但不像容器、算法，直到现在，仍然有且只有这么一个字符串类，“只此一家，别无分号”。

所以，在这种“别无选择”的情况下，我们就要多了解它的优缺点，尽量用好它。

首先你要看到，`string` 是一个功能比较齐全的字符串类，可以提取子串、比较大小、检查长度、搜索字符.....基本满足一般人对字符串的“想象”。

 复制代码

```
1 string str = "abc";
2
3 assert(str.length() == 3);
```

```
4 assert(str < "xyz");
5 assert(str.substr(0, 1) == "a");
6 assert(str[1] == 'b');
7 assert(str.find("1") == string::npos);
8
```

刚才也说了，string 的接口比较复杂，除了字符串操作，还有 size()、begin()、end()、push_back() 等类似容器的操作，这很容易让人产生“联想”，把它当成是一个“字符容器”。

但我不建议你这样做。**字符串和容器完全是两个不同的概念。**

字符串是“文本”，里面的字符之间是强关系，顺序不能随便调换，否则就失去了意义，通常应该视为一个整体来处理。而容器是“集合”，里面的元素之间没有任何关系，可以随意增删改，对容器更多地是操作里面的单个元素。

理解了这一点，**把每个字符串都看作是一个不可变的实体，你才能在 C++ 里真正地用好字符串。**

但有的时候，我们也确实需要存储字符的容器，比如字节序列、数据缓冲区，这该怎么办呢？

这个时候，我建议你**最好改用**vector<char>，它的含义十分“纯粹”，只存储字符，没有 string 那些不必要的成本，用起来也就更灵活一些。

接下来我们再看看 string 的一些小技巧。

1. 字面量后缀

C++11 为方便使用字符串，新增了一个字面量的**后缀“s”**，明确地表示它是 string 字符串类型，而不是 C 字符串，这就可以利用 auto 来自动类型推导，而且在其他用到字符串的地方，也可以省去声明临时字符串变量的麻烦，效率也会更高：

```
1 using namespace std::literals::string_literals; //必须打开名字空间
2
```

 复制代码

```
3 auto str = "std string"s;    // 后缀s, 表示是标准字符串, 直接类型推导
4
5
```

不过要提醒你的是，**为了避免与用户自定义字面量的冲突，后缀“s”不能直接使用，必须用 using 打开名字空间才行**，这是它的一个小缺点。

2. 原始字符串

C++11 还为字面量增加了一个**“原始字符串”**（Raw string literal）的新表示形式，比原来的引号多了一个大写字母 R 和一对圆括号，就像下面这样：

```
1 auto str = R"(nier:automata)";    // 原始字符串: nier:automata
```

 复制代码

这种形式初看上去显得有点多余，它有什么好处呢？


你一定知道，C++ 的字符有“转义”的用法，在字符前面加上一个“\”，就可以写出“\n”“\t”来表示回车、跳格等不可打印字符。

但这个特性也会带来麻烦，有时我们不想转义，只想要字符串的“原始”形式，在 C++ 里写起来就很难受了。特别是在用正则表达式的时候，由于它也有转义，两个转义效果“相乘”，就很容易出错。

比如说，我要在正则里表示“\”，需要写成“\\”，而在 C++ 里需要对“\”再次转义，就是“\\\\”，你能数出来里面到底有多少个“\”吗？

如果使用原始字符串的话，就没有这样的烦恼了，它不会对字符串里的内容做任何转义，完全保持了“原始风貌”，即使里面有再多的特殊字符都不怕：


```
1 auto str1 = R"(char"'"')";    // 原样输出: char"'"'
2 auto str2 = R"(\r\n\t)";      // 原样输出: \r\n\t
3 auto str3 = R"(\\$)";         // 原样输出: \\$
4 auto str4 = "\\\\\\$";        // 转义后输出: \\$
```

 复制代码

不过，想要在原始字符串里面写引号 + 圆括号的形式该怎么办呢？

对于这个问题，C++ 也准备了应对的办法，就是在圆括号的两边加上最多 16 个字符的特别“界定符”（delimiter），这样就能够保证不与字符串内容发生冲突：

```
1 auto str5 = R"==(R"(xxx))==";// 原样输出: R"(xxx)"
```

 复制代码

3. 字符串转换函数

在处理字符串的时候，我们还会经常遇到与数字互相转换的事情，以前只能用 C 函数 `atoi()`、`atol()`，它们的参数是 C 字符串而不是 `string`，用起来就比较麻烦，于是，C++11 就增加了几个新的转换函数：

`stoi()`、`stol()`、`stoll()` 等把字符串转换成整数；

`stof()`、`stod()` 等把字符串转换成浮点数；

`to_string()` 把整数、浮点数转换成字符串。

这几个小函数在处理用户数据、输入输出的时候，非常方便：

```
1 assert(stoi("42") == 42);           // 字符串转整数
2 assert(stol("253") == 253L);        // 字符串转长整数
3 assert(stod("2.0") == 2.0);         // 字符串转浮点数
4
5 assert(to_string(1984) == "1984");   // 整数转字符串
```


 复制代码

4. 字符串视图类

再来说一下 `string` 的成本问题。它确实有点“重”，大字符串的拷贝、修改代价很高，所以我们通常都尽量用 `const string&`，但有的时候还是无法避免（比如使用 C 字符串、获取子串）。如果你对此很在意，就有必要找一个“轻量级”的替代品。

在 C++17 里，就有这么一个完美满足所有需求的东西，叫 `string_view`。顾名思义，它是一个字符串的视图，成本很低，内部只保存一个指针和长度，无论是拷贝，还是修改，都非常廉价。

唯一的遗憾是，它只出现在 C++17 里，不过你也可以参考它的接口，自己在 C++11 里实现一个简化版本。下面我给你一个简单的示范，你可以课下去扩展：

 复制代码

```
1  class my_string_view final          // 简单的字符串视图类，示范实现
2  {
3  public:
4      using this_type = my_string_view;    // 各种内部类型定义
5      using string_type = std::string;
6      using string_ref_type = const std::string&;
7
8      using char_ptr_type = const char*;
9      using size_type = size_t;
10 private:
11     char_ptr_type ptr = nullptr;        // 字符串指针
12     size_type len = 0;                  // 字符串长度
13 public:
14     my_string_view() = default;
15     ~my_string_view() = default;
16
17     my_string_view(string_ref_type str) noexcept
18         : ptr(str.data()), len(str.length())
19     {}
20 public:
21     char_ptr_type data() const          // 常函数，返回字符串指针
22     {
23         return ptr;
24     }
25
26     size_type size() const              // 常函数，返回字符串长度
27     {
28         return len;
29     }
30 };
```

正则表达式

说了大半天，其实我们还是没有回答这节课开头提出的疑问，也就是“在 C++ 里该怎么处理文本”。`string` 只是解决了文本的表示和存储问题，要对它做大小写转换、判断前缀后缀、模式匹配查找等更复杂的处理，要如何做呢？

使用标准算法显然是不行的，因为算法的工作对象是容器，而刚才我就说了，字符串与容器是两个完全不同的东西，大部分算法都无法直接套用到字符串上，所以文本处理也一直是 C++ 的“软肋”。

好在 C++11 终于在标准库里加入了正则表达式库 regex（虽然有点晚），利用它的强大能力，你就能够任意操作文本、字符串。

很多语言都支持正则表达式，关于它的语法规则我也就不细说了（课下你可以参考下这个链接：[@https://www.pcre.org/](https://www.pcre.org/)），我就重点介绍一下在 C++ 里怎么用。

C++ 正则表达式主要有两个类。

regex：表示一个正则表达式，是 basic_regex 的特化形式；

smatch：表示正则表达式的匹配结果，是 match_results 的特化形式。

C++ 正则匹配有三个算法，注意它们都是“只读”的，不会变动原字符串。

regex_match()：完全匹配一个字符串；

regex_search()：在字符串里查找一个正则匹配；

regex_replace()：正则查找再做替换。

所以，你只要用 regex 定义好一个表达式，然后再调用匹配算法，就可以立刻得到结果，用起来和其他语言差不多。不过，在写正则的时候，记得最好要用“原始字符串”，不然转义符绝对会把你折腾得够呛。

下面我举个例子：

```
1  auto make_regex = [](const auto& txt)    // 生产正则表达式
2  {
3      return std::regex(txt);
4  };
5
6  auto make_match = []()                  // 生产正则匹配结果
7  {
8      return std::smatch();
```

 复制代码



```

9  };
10
11  auto str = "neir:automata"s;           // 待匹配的字符串
12  auto reg =
13      make_regex(R"^(^(\w+)\: (\w+)\$)"); // 原始字符串定义正则表达式
14

```

这里我先定义了两个简单的 lambda 表达式，生产正则对象，主要是为了方便用 auto 自动类型推导。当然，同时也隐藏了具体的类型信息，将来可以随时变化（这也有点函数式编程的味道了）。

然后我们就可以调用 `regex_match()` 检查字符串，函数会返回 `bool` 值表示是否完全匹配正则。如果匹配成功，结果存储在 `what` 里，可以像容器那样去访问，第 0 号元素是整个匹配串，其他的是子表达式匹配串：

 复制代码

```

1  assert(regex_match(str, what, reg)); // 正则匹配
2
3  for(const auto& x : what) {           // for遍历匹配的子表达式
4      cout << x << ',';
5  }

```

`regex_search()`、`regex_replace()` 的用法也都差不多，很好理解，直接看代码吧：

 复制代码

```

1  auto str = "god of war"s;           // 待匹配的字符串
2
3  auto reg =
4      make_regex(R"((\w+)\s(\w+))"); // 原始字符串定义正则表达式
5  auto what = make_match();           // 准备获取匹配的结果
6
7  auto found = regex_search(           // 正则查找，和匹配类似
8      str, what, reg);
9
10 assert(found);                       // 断言找到匹配
11 assert(!what.empty());               // 断言有匹配结果
12 assert(what[1] == "god");           // 看第一个子表达式
13 assert(what[2] == "of");            // 看第二个子表达式
14
15 auto new_str = regex_replace(         // 正则替换，返回新字符串
16     str,                               // 原字符串不改动
17     make_regex(R"(\w+\$)"),           // 就地生成正则表达式对象
18     "peace"                           // 需要指定替换的文字

```

```
19 );  
20  
21 cout << new str << endl;           // 输出god of peace
```

这段代码的 `regex_search()` 搜索了两个连续的单词，然后在匹配结果里以数组下标的形式输出。

`regex_replace()` 不需要匹配结果，而是要提供一个替换字符串，因为算法是“只读”的，所以它会返回修改后的新字符串。利用这一点，就可以把它的输出作为另一个函数的输入，用“函数套函数”的形式实现“函数式编程”。

在使用 `regex` 的时候，还要注意正则表达式的成本。因为正则串只有在运行时才会处理，检查语法、编译成正则对象的代价很高，所以**尽量不要反复创建正则对象，能重用就重用**。在使用循环的时候更要特别注意，一定要把正则提到循环体外。

`regex` 库的功能非常强大，我们没有办法把方方面面的内容都涉及到，刚刚我讲的都是最实用的方法。像大小写敏感、优化匹配引擎、扩展语法、正则迭代 / 切分等其他高级的功能，建议你课下多努力，参考一下 [🔗 GitHub](#) 仓库里的资料链接，深入研究它的接口和设置参数。

小结

好了，今天我讲了字符串类 `string` 和正则表达式库 `regex`，它们是 C++ 标准库里处理文本的唯一工具，虽然离完美还有距离，但我们也别无选择。目前我们能做的，就是充分掌握一些核心技巧，规避一些使用误区。这节课是我的经验总结，建议你多读几遍，希望可以进一步提升你的编码能力。

简单小结一下今天的内容：

1. C++ 支持多种字符类型，常用的 `string` 其实是模板类 `basic_string` 的特化形式；
2. 目前 C++ 对 Unicode 的支持还不太完善，建议尽量避开国际化和编码转化，不要“自讨苦吃”；
3. 应当把 `string` 视为一个完整的字符串来操作，不要把它当成容器来使用；
4. 字面量后缀 `"s"` 表示字符串类，可以用来自动推导出 `string` 类型；

5. 原始字符串不会转义，是字符串的原始形态，适合在代码里写复杂的文本；
6. 处理文本应当使用正则表达式库 `regex`，它的功能非常强大，但需要花一些时间和精力才能掌握。

课下作业

最后是课下作业时间，给你留两个思考题：

1. 你平时在使用字符串的时候有感觉到哪些不方便吗？如果有的话，是怎么解决的？
2. 你觉得正则表达式能够应用在什么地方，解决哪些实际的问题？

欢迎你在留言区写下你的思考和答案，如果觉得今天的内容对你有所帮助，也欢迎分享给你的朋友。我们下节课见。

课外小贴士

1. C++20新增了`char8_t`，它相当于`unsigned char`，专门用来表示UTF-8编码，相应的也就有专门的字符串类`u8string`和字面量前缀`u8`。
2. 在`string`转换C字符串的时候，需要注意`c_str()`与`data()`的区别，两个函数都返回`const char*`指针，但`c_str()`会在末尾添加一个`'\0'`。
3. Boost程序库里有一个工具`lexical_cast`，可以在字符串和数字之间互转，功能比

stoi()、to_string()更强大。

4. basic_regex、match_results针对char*、string有不同的特化形式，但命名上却不太一致，有cmatch、smatch，却没有对应的cregex、sregex，我个人建议使用类型别名来保持对应关系，看起来更清楚。
5. 除了标准库里的regex库，你也可以选择其他的第三方正则表达式库，比如PCRE、Hyperscan、libsregex。



课程预告

6月-7月课表抢先看

充 ¥500 得 ¥580

赠「¥ 118 月球主题 AR 笔记本」



【点击】图片，立即查看 >>>

精选留言 (6)

写留言



Weining Cao

2020-05-30

处理string的话，标准C++的接口实在不够友好，易用。反而第三方库QT的QString用起来就舒服，顺手很多。

在这方面，python语言的string处理比C++要友好太多。比如最简单的string内子串替换功能，python可以直接`str.replace(a, b)`，但C++的`replace`函数需要先计算出替换s...
展开

作者回复: 现在有了regex好了一点，用boost里的string_algo也比较方便。



this_is_for_u

2020-05-30

关于`.data`和`.c_str`得区别，都说是多了个`\0`，可我看gcc9.2的源码发现它俩没有任何区别啊，都没有`\0`，不知道自己哪里看错了！

展开

作者回复: 看gcc源码，很厉害啊。

这里说的标准里的规定，`c_str()`必须有`\0`结尾，而`data()`则不保证有`\0`，可以实际写代码试试。



EncodedStar

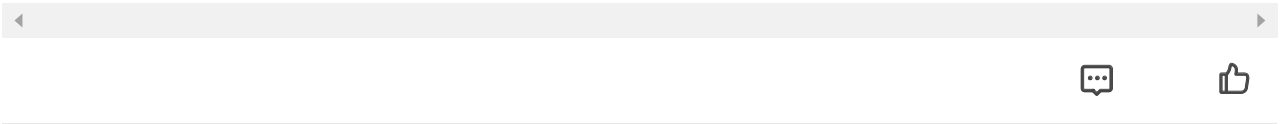
2020-05-30

遇到转换类型int转换string或者string转换int时确实觉得麻烦，之前版本没啥好的库函数，都是用`sstream`来流，或者有一些`atoi`, `itoa`, `c11`又出来`stoi`(看来前人也是遇到了麻烦)等来进行处理。语言迭代之所以更新，在处理时也是因为遇到的问题比较多，为了更方便而迭代起来。

正则一般都是用来匹配文件过传过来消息时有效字段时候用的。

展开

作者回复: regex还是有点重, 如果什么都用正则也是挺麻烦的, 所以我觉得C++在这方面还应该加强一下, 让字符串用起来更方便些。

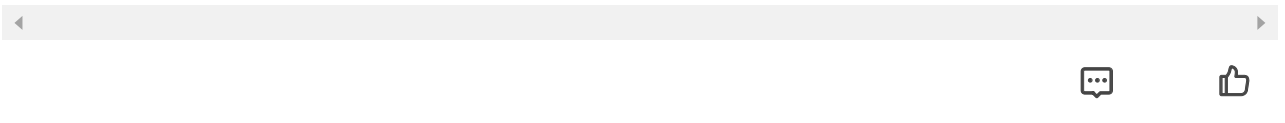


SometimesNever

2020-05-30

老师, 我想问个之前的智能指针相关的, 什么情况不能用智能指针代替裸指针呢?

作者回复: 对内存占用和效率很敏感, 或者必须手工精细管理资源。



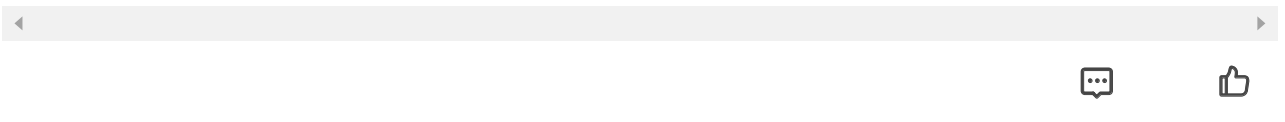
桐仲康

2020-05-30

请问老师, 标准库里自带的stoi函数效率如何呢?

展开 ▾

作者回复: 这个没试过, 可以自己写点代码, 跑个几十万次测试一下, 不过这个函数一般也不会对性能有太大要求吧。



TC128

2020-05-30

老师, 小结2说尽量不要用UNICODE, 但如果用C++写界面 (MFC、DirectUI), 且软件又需要国际化, 这种情况也尽量不用UNICODE吗? 还是说换个语言写界面?

作者回复: 什么事情都不是绝对的, 像这种情况就必须用Unicode, 但用C++处理还是挺费劲的, 其他语言做起来要比C++容易一些。

